

# LL Grammars

Mattox Beckman

`beckman@iit.edu`

Illinois Institute of Technology

The topic for this lecture is a kind of grammar that works well with recursive-descent parsing.

- Know how to tell if a grammar is LL.
- Know what parsing technique will work with an LL grammar.
- Know how to detect and eliminate left recursion.
- Know how to detect and eliminate common prefixes.

Further reading: See Dragon Book §4.x

# What is LL(n) Parsing?

## §0 Objectives

---

- An LL parse uses a **L**eft-to-right scan and produces a **L**eftmost derivation, using **n** tokens of lookahead.
- A.K.A. Top-Down Parsing

Example Grammar:

$$S \rightarrow + E E$$

$$E \rightarrow \text{int}$$

$$E \rightarrow * E E$$

Example Input: + 2 \* 3 4

Syntax Tree:

S

Derivations:

S

# What is LL(n) Parsing?

## §0 Objectives

- An LL parse uses a **L**eft-to-right scan and produces a **L**eftmost derivation, using **n** tokens of lookahead.
- A.K.A. Top-Down Parsing

Example Grammar:

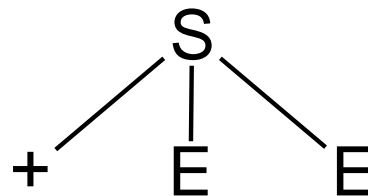
$$S \rightarrow + E E$$

$$E \rightarrow \text{int}$$

$$E \rightarrow * E E$$

Example Input: + 2 \* 3 4

Syntax Tree:



Derivations:

S

+ E E

# What is LL(n) Parsing?

## §0 Objectives

- An LL parse uses a **L**eft-to-right scan and produces a **L**eftmost derivation, using **n** tokens of lookahead.
- A.K.A. Top-Down Parsing

Example Grammar:

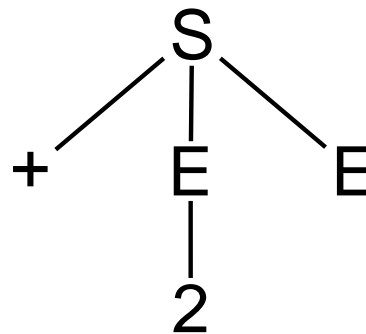
$$S \rightarrow + E E$$

$$E \rightarrow \text{int}$$

$$E \rightarrow * E E$$

Example Input: + 2 \* 3 4

Syntax Tree:



Derivations:

S

+ E E

+ 2 E

# What is LL(n) Parsing?

## §0 Objectives

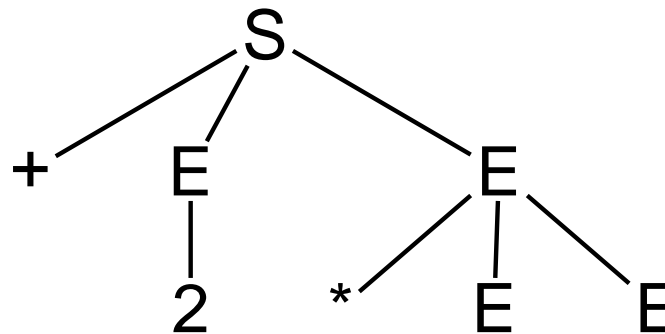
- An LL parse uses a **Left-to-right** scan and produces a **Leftmost** derivation, using **n** tokens of lookahead.
- A.K.A. Top-Down Parsing

Example Grammar:

$$S \rightarrow + E E$$
$$E \rightarrow \text{int}$$
$$E \rightarrow * E E$$

Example Input: + 2 \* 3 4

Syntax Tree:



Derivations:

S

+ E E

+ 2 E

+ 2 \* E E

# What is LL(n) Parsing?

## §0 Objectives

- An LL parse uses a **Left-to-right** scan and produces a **Leftmost** derivation, using **n** tokens of lookahead.
- A.K.A. Top-Down Parsing

Example Grammar:

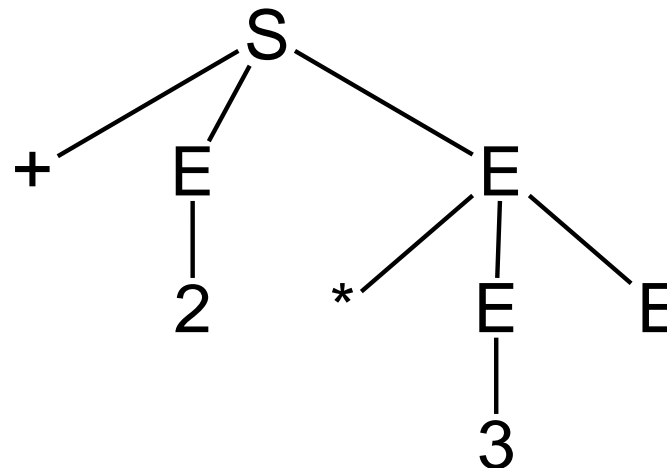
$$S \rightarrow + E E$$

$$E \rightarrow \text{int}$$

$$E \rightarrow * E E$$

Example Input: + 2 \* 3 4

Syntax Tree:



Derivations:

S  
 + E E  
 + 2 E  
 + 2 \* E E  
 + 2 \* 3 E

# What is LL(n) Parsing?

## §0 Objectives

- An LL parse uses a **Left-to-right** scan and produces a **Leftmost** derivation, using **n** tokens of lookahead.
- A.K.A. Top-Down Parsing

Example Grammar:

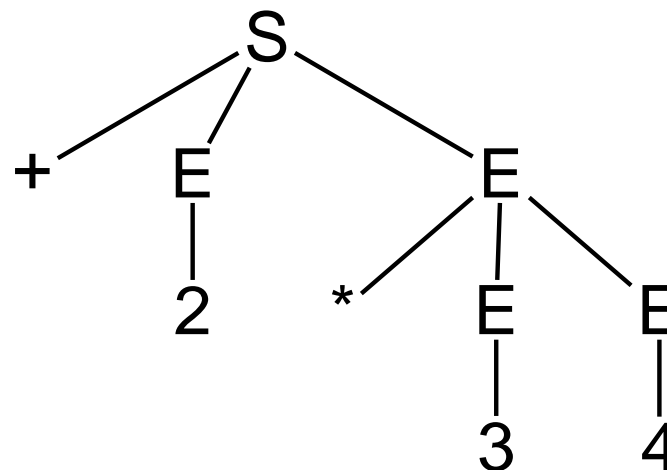
$$S \rightarrow + E E$$

$$E \rightarrow \text{int}$$

$$E \rightarrow * E E$$

Example Input: + 2 \* 3 4

Syntax Tree:



Derivations:

S				
+	E	E		
+	2	E		
+	2	*	E	E
+	2	*	3	E
+	2	*	3	4



# How to Implement It

## §0 Objectives

---

LL Parsers are very easy to implement.

- Consider each non-terminal rule to be a function.
- Each terminal consumes an input.
- Each function has type `string list -> tree * string list`
  - input is a list of tokens
  - output is a syntax tree and remaining tokens.
- You also need to create a type to represent your tree.

```
1 # getS ["+"; "2"; "*"; "3"; "4"];;  
2 - : s * string list =  
3   S (Eint 2, Etimes (Eint 3, Eint 4)), []
```

- Each function immediately checks the first token of the input string. (LL(0) parsers will consume the token immediately.)
- This token is used to decide what to do next.

```

1 let rec getE input =
2   match input with
3   | [] -> raise ParseError
4   | "*"::xs -> let (e1,rest) = getE xs in
5                 let (e2,rest) = getE rest in
6                 (Etimes (e1,e2), rest)
7   | i::xs -> (Eint (int_of_string i), xs)
```

- What kinds of things could go wrong?

- A rule like  $E \rightarrow E + E$  would cause an infinite loop.

$E \rightarrow E + E$

```
1 let rec getE input =  
2   match input with  
3   | [] -> raise ParseError  
4   | _ -> let (e1,rest) = getE input in  
5           let ["+";rest] = rest in  
6           let (e2,rest) = getE rest in  
7           (Eplus (e1,e2), rest)
```

## Rules with Common Prefixes

## §1 Breaking LL Parsers

- A rule like  $E \rightarrow - E \mid - E E$  would confuse the function.  
Which version of the rule should be used?

```
1 let rec getE input =  
2   match input with  
3   | [] -> raise ParseError  
4   | "-"::xs -> let (e1,rest) = getE xs in  
5                 (Enegative e1, rest)  
6   | "-"::xs -> let (e1,rest) = getE xs in  
7                 let (e2,rest) = getE rest in  
8                 (Eminus (e1,e2), rest)
```

**The Idea****§2 Eliminating Left Recursion**

---

Consider deriving  $i++++$  from the following grammar:

$E$

Derivation:

$E$

$$E \rightarrow E +$$

$$E \rightarrow i$$

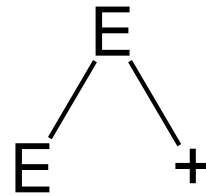
**The Idea****§2 Eliminating Left Recursion**

---

Consider deriving  $i++++$  from the following grammar:

$$E \rightarrow E +$$

$$E \rightarrow i$$



Derivation:

$E$

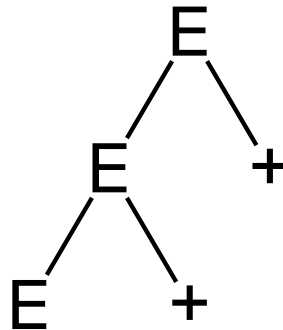
$E+$

## The Idea

## §2 Eliminating Left Recursion

---

Consider deriving  $i++++$  from the following grammar:

$$E \rightarrow E +$$
$$E \rightarrow i$$


Derivation:

$E$

$E +$

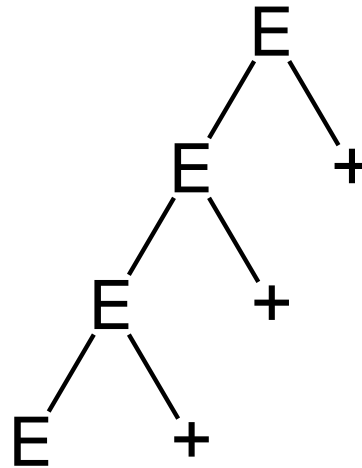
$E ++$

# The Idea

## §2 Eliminating Left Recursion

---

Consider deriving  $i++++$  from the following grammar:

$$E \rightarrow E +$$
$$E \rightarrow i$$


Derivation:

$E$

$E +$

$E ++$

$E +++$



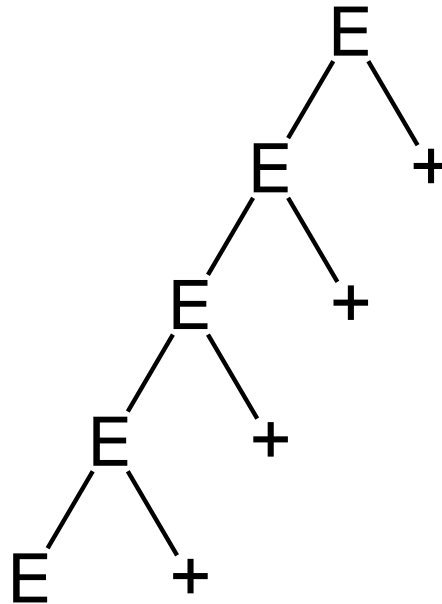
## The Idea

## §2 Eliminating Left Recursion

Consider deriving  $i++++$  from the following grammar:

$$E \rightarrow E +$$

$$E \rightarrow i$$



Derivation:

$E$

$E +$

$E ++$

$E +++$

$E ++++$

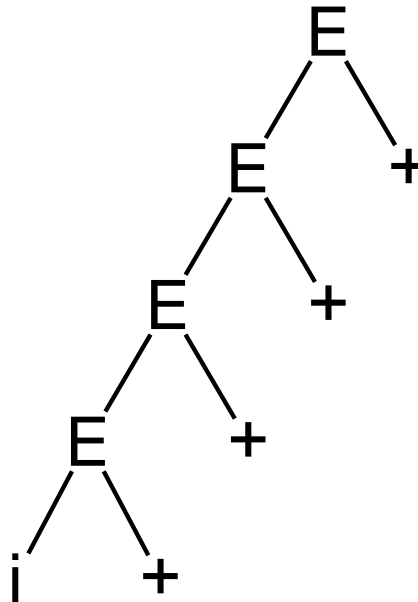
## The Idea

## §2 Eliminating Left Recursion

Consider deriving  $i++++$  from the following grammar:

$$E \rightarrow E +$$

$$E \rightarrow i$$



Derivation:

$E$

$E +$

$E ++$

$E +++$

$E ++++$

$i++++$

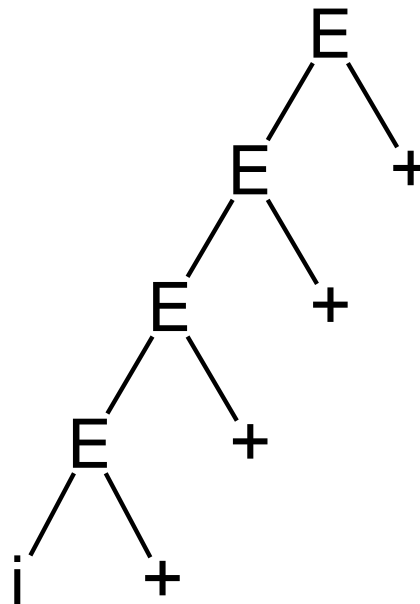
## The Idea

## §2 Eliminating Left Recursion

Consider deriving  $i++++$  from the following grammar:

$$E \rightarrow E +$$

$$E \rightarrow i$$



Derivation:

$E$

$E +$

$E ++$

$E +++$

$E ++++$

$i++++$

- The rule  $E \rightarrow E +$  says that we can have as many  $+$ s as we want *at the end* of the sentence.
- The rule  $E \rightarrow i$  says—in effect—that the first word can be a  $i$ .
- Question: isn't there another way to write this?

**More complicated example****§2 Eliminating Left Recursion**

---

Consider the following grammar. What does it mean?

$$B \rightarrow Bxy \mid Bz \mid q \mid r$$

- At the end can come any combination of  $x$   $y$  or  $z$
- At the beginning can come  $q$  or  $r$

# Eliminating the Left Recursion

## §2 Eliminating Left Recursion

We can rewrite these grammars

$$E \rightarrow E + \mid i$$

$$B \rightarrow Bxy \mid Bz \mid q \mid r$$

using the following transformation:

- Productions of the form  $S \rightarrow \beta$  become  $S \rightarrow \beta S'$ .
- Productions of the form  $S \rightarrow S\alpha$  become  $S' \rightarrow \alpha S'$ .
- Add  $S' \rightarrow \epsilon$ .

Result:

$$E \rightarrow iE'$$

$$E' \rightarrow +E' \mid \epsilon$$

$$B \rightarrow qB' \mid rB'$$

$$B' \rightarrow xyB' \mid zB' \mid \epsilon$$

# Mutual Recursions!

## §2 Eliminating Left Recursion

---

Things are slightly more complicated if we have mutual recursions.

$$\begin{aligned}A &\rightarrow Aa \mid Bb \mid Cc \mid q \\B &\rightarrow Ax \mid By \mid Cz \mid rA \\C &\rightarrow Ai \mid Bj \mid Ck \mid sB\end{aligned}$$

How to do it:

- Take the first symbol (A) and eliminate immediate left recursion.
- Take the second symbol (B), and substitute left recursions to A. Then eliminate immediate left recursion in B.
- Take the third symbol (C) and substitute left recursions to A and B. Then eliminate immediate left recursion in C.

## Left Recursion Example

## §2 Eliminating Left Recursion

---

Here is a more complex left recursion.

$$A \rightarrow Aa \mid Bb \mid Cc \mid q$$

$$B \rightarrow Ax \mid By \mid Cz \mid rA$$

$$C \rightarrow Ai \mid Bj \mid Ck \mid sB$$

First we eliminate the left recursion from  $A$ .

$$A \rightarrow Aa \mid Bb \mid Cc \mid q$$

becomes

$$A \rightarrow BbA' \mid CcA' \mid qA'$$

$$A' \rightarrow aA' \mid \epsilon$$

## Left Recursion Example, 2

## §2 Eliminating Left Recursion

---

We substituting in the new definition of  $A$ , and now we will work on the  $B$  productions.

$$A \rightarrow BbA' \mid CcA' \mid qA'$$

$$A' \rightarrow aA' \mid \epsilon$$

$$B \rightarrow Ax \mid By \mid Cz \mid rA$$

$$C \rightarrow Ai \mid Bj \mid Ck \mid sB$$

First, we eliminate the “backward” recursion from  $B$  to  $A$ .

$B \rightarrow Ax$  becomes

$$B \rightarrow BbA'x \mid CcA'x \mid qA'x$$



**Left Recursion Example, 3****§2 Eliminating Left Recursion**

---

$$A \rightarrow BbA' \mid CcA' \mid qA'$$

$$A' \rightarrow aA' \mid \epsilon$$

$$B \rightarrow BbA'x \mid CcA'x \mid qA'x \mid By \mid Cz \mid rA$$

$$C \rightarrow Ai \mid Bj \mid Ck \mid sB$$

Now we can eliminate the simple left recursion in  $B$ , to get

$$B \rightarrow CcA'xB' \mid qA'xB' \mid CzB' \mid rAB'$$

$$B' \rightarrow bA'xB' \mid yB' \mid \epsilon$$

**Left Recursion Example, 4****§2 Eliminating Left Recursion**

$$A \rightarrow BbA' \mid CcA' \mid qA'$$

$$A' \rightarrow aA' \mid \epsilon$$

$$B \rightarrow CcA'xB' \mid qA'xB' \mid CzB' \mid rAB'$$

$$B' \rightarrow bA'xB' \mid yB' \mid \epsilon$$

$$C \rightarrow Ai \mid Bj \mid Ck \mid sB$$

Now production  $C$ : first, replace left recursive calls to  $A$ ...

$$C \rightarrow \boxed{B}bA'i \mid CcA'i \mid qA'i \mid \boxed{B}j \mid Ck \mid sB$$

Next, replace left recursive calls to  $B$  (this gets messy)...

$$C \rightarrow \boxed{CcA'xB'}bA'i \mid \boxed{qA'xB'}bA'i \mid \boxed{CzB'}bA'i \mid \boxed{rAB'}bA'i$$

$$\boxed{CcA'xB'}j \mid \boxed{qA'xB'}j \mid \boxed{CzB'}j \mid \boxed{rAB'}j$$

$$CcA'i \mid qA'i \mid Ck \mid sB$$

**Left Recursion Example, 5****§2 Eliminating Left Recursion**

---

Reorganizing  $C$ , we have

$$C \rightarrow qA'xB'bA'i \mid rAB'bA'i \mid qA'xB'j \mid rAB'j \mid qA'i \mid sB$$

$$CcA'xB'bA'i \mid CzB'bA'i \mid CcA'xB'j \mid CzB'j \mid CcA'i \mid Ck$$

Eliminating left recursion gives us

$$C \rightarrow qA'xB'bA'iC' \mid rAB'bA'iC' \mid qA'xB'jC' \mid rAB'jC' \mid qA'iC' \mid sBC$$

$$C' \rightarrow cA'xB'bA'iC' \mid zB'bA'iC' \mid cA'xB'jC' \mid zB'jC' \mid cA'iC' \mid kC' \mid \epsilon$$

Our final grammar is now

$$A \rightarrow BbA' \mid CcA' \mid qA'$$

$$A' \rightarrow aA' \mid \epsilon$$

$$B \rightarrow CcA'xB' \mid qA'xB' \mid CzB' \mid rAB'$$

$$B' \rightarrow bA'xB' \mid yB' \mid \epsilon$$

$$C \rightarrow qA'xB'bA'iC' \mid rAB'bA'iC' \mid qA'xB'jC' \mid rAB'jC' \mid qA'iC' \mid sBC'$$

$$C' \rightarrow cA'xB'bA'iC' \mid zB'bA'iC' \mid cA'xB'jC' \mid zB'jC' \mid cA'iC' \mid kC' \mid \epsilon$$

Beautiful, isn't it? I wonder why we don't do this more often?

- Disclaimer: if there is a cycle ( $A \rightarrow^+ A$ ) or an epsilon production ( $A \rightarrow \epsilon$ ) then this technique is not guaranteed to work.

This grammar has common prefixes.

$$\begin{aligned}A &\rightarrow xyB \mid CyC \mid q \\B &\rightarrow zC \mid zx \mid w \\C &\rightarrow y \mid x\end{aligned}$$

To check for common prefixes, take a non-terminal and compare the First sets of each production.

Production	FirstSet	If we are viewing an $A$ , we will want to look at the next token to see which $A$ production to use. If that token is $x$ , then which production do we use?
$A \rightarrow xyB$	$\{x\}$	
$A \rightarrow CyC$	$\{x, y\}$	
$A \rightarrow q$	$\{q\}$	

If  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \gamma$  we can rewrite it as

$$\begin{array}{l} A \rightarrow \alpha A' \mid \gamma \\ A' \rightarrow \beta_1 \mid \beta_2 \end{array}$$

So, in our example:

$A \rightarrow xyB \mid CyC \mid q$	becomes	$A \rightarrow xA' \mid q \mid yyC$
$B \rightarrow zC \mid zx \mid w$		$A' \rightarrow yB \mid yC$
$C \rightarrow y \mid x$		$B \rightarrow zB' \mid w$
		$B' \rightarrow C \mid x$
		$C \rightarrow y \mid x$

Sometimes you'll need to do this more than once. Note that this process can destroy the meaning of the nonterminals.

One of these is LL, the other two are not. Fix the ones that are not.

**Grammar 1**

$$E \rightarrow E \ x \ y$$

$$E \rightarrow E \ x \ B$$

$$E \rightarrow q$$

$$B \rightarrow E \ z$$

**Grammar 2**

$$S \rightarrow A \ x$$

$$S \rightarrow B \ y$$

$$A \rightarrow z \ B$$

$$B \rightarrow w \ A$$

**Grammar 3**

$$S \rightarrow A \ x$$

$$S \rightarrow B \ y$$

$$A \rightarrow z \ B$$

$$B \rightarrow z$$

Grammar 1 starts as:      Eliminate the left-recursion to get:      Fix the common prefixes to get....

$$E \rightarrow E \ x \ y$$

$$E \rightarrow E \ x \ B$$

$$E \rightarrow q$$

$$B \rightarrow E \ z$$

$$E \rightarrow q E'$$

$$E' \rightarrow x \ y \ E'$$

$$E' \rightarrow x \ B \ E'$$

$$E' \rightarrow \epsilon$$

$$B \rightarrow E \ z$$

$$E \rightarrow q E'$$

$$E' \rightarrow x \ E''$$

$$E' \rightarrow \epsilon$$

$$E'' \rightarrow y \ E' \mid B \ E'$$

$$B \rightarrow E \ z$$



$$S \rightarrow A x$$

$$S \rightarrow B y$$

$$A \rightarrow z B$$

$$B \rightarrow w A$$

It doesn't terminate. But it's not left recursive, and it has no productions with common prefixes, so it's still LL.

$$S \rightarrow A \ x$$

$$S \rightarrow B \ y$$

$$A \rightarrow z \ B$$

$$B \rightarrow z$$

Production  $S$  has common prefixes. One way to fix this grammar is to eliminate the distinctions between  $A$  and  $B$  — this may not be what you want, though.

$$S \rightarrow z \ S'$$

$$S' \rightarrow z \ x$$

$$S' \rightarrow y$$